Hybrid Event Store

Editor: David Adams

Contributors: David Adams, Wensheng Deng, Valeri Fine, Yuri Fisyak, Hong Ma, Pavel Nevski, Victor Pervoztchikov, Srini Rajagopalan, Torre Wenaus

We thank the following for comments and suggestions: D. Malon, D. Quarrie, R. D. Schaffer and P. Calafiura, A. Vaniachine, S. Eckmann.

Version 1.5.3

February 28, 2002 10:00 AM EST

This document is preliminary.

The following are not yet included.

1. *Generic data access (inheritance)*
2. *EDO histories (including parent and reference EDO's)*
3. *HesPcReference file format (supports only PC refs; no PC by value)*
4. *Support for overwriting a visible EDO specified by type-key. This may be a bad idea.*

We plan make the following additions and changes next:

1. *Want to dump all data to persistent store. This is probably a special stream type. Call it ALL.*

1. Introduction

The architecture of the ATLAS event store is described in a separate document[2]. Here we present a realization of that architecture in which a relational database is used to keep track of files that store and provide access to the objects that make up the event data. This is called the hybrid event store.

We include a description of our interpretation of the architecture to resolve ambiguities and to enable the reader to follow our discussion without prerequisite.

We begin with a glossary where we have tried to carry over the ideas and terminology of the architecture design document. We then identify the components of the model and provide use cases, requirements and some of the design for each component.

2. Definitions

The following define the language used to describe the model and also go a long ways toward describing the model itself.

Event data

High energy physics event data is that associated with particular beam crossing.

Beam crossing

Data is acquired at each triggered beam crossing or collision.

Event

The word event is used in many ways. It can refer to a beam crossing or to a subset data associated with a particular crossing. The latter might include the data in a file or that visible during event processing.

Event ID

Each triggered beam crossing is assigned a unique identifier called the event ID. These identifiers are ordered (most likely temporally) so that we can speak of a range of event ID's.

Raw data

The raw data is the event data read directly from the detector. All other event data is derived from the raw data corresponding to the same crossing. There is only a weak coupling (though global data) to data from other beam crossings.

Monte Carlo data

Data generated from simulation rather than actual detector readout is known as Monte Carlo data. In this case the source of the data is the random number seeds, i.e. the states of the random number generators. Simulated raw data may be produced.

Global data

We use the term global data to refer to that which is associated with multiple beam crossings. It includes information about the geometry and alignment of the detector and the calibration data used to connect the raw data to physical quantities such as spatial positions or energy depositions. This data is obtained from non-event sources and by combining information from multiple beam crossings.

Reconstruction

The reconstruction of event data is the process of using existing data from a beam crossing to produce new data associated with that same beam crossing. For example raw data may be used to produce clusters that are used to find tracks that are used to create electrons. Global data may also be used as input in some stages of reconstruction.

Algorithm

Reconstruction is carried out in a series of well-defined steps. One algorithm is applied at each step. Each algorithm takes a subset of the existing event data and possibly some global data as input.

An algorithm may also act as a selector where it examines the data from the current beam crossing and returns a status that can be used to decide if the data should be recorded.

Algorithm instance

The algorithm is characterized by type, version, release version, run-time parameters and name. A unique combination of these is called an algorithm instance. This is sometimes abbreviated as algorithm.

Algorithm ID

Any algorithm instance used in official production is assigned a unique identifier called the algorithm ID.

Event data object (EDO)

In our object-oriented view of the world it is natural to describe the data from a particular beam crossing as a collection of objects. Rather than attempt to keep track of low-level objects such as individual tracks or electrons, relational databases are used to keep track of files that, in turn, manage collections of these low-level objects. We refer to these collections as event data objects or EDO's. These are called DataObjects in StoreGate.

An algorithm takes one or more EDO's as input and typically produces a single EDO as output.

Parent EDO

The EDO's used as input when creating an EDO are called the parents.

Referenced EDO

In the context of an EDO, a referenced EDO is another (or the same) EDO that contains objects that are referenced from object in the first EDO. A referenced EDO may be the current one, a parent or one from any previous generation.

EDO key

EDO's are characterized by their type (e.g. collection of tracks vs. collection of jets) but it is sometimes necessary to distinguish between EDO's of the same type. For example we might want to distinguish jets found with different algorithm types or with different cone sizes. This distinction can be made by keeping an association with the algorithm (type, version and parameters) that produced the EDO but this is cumbersome. Also, there may be occasions when we want to label EDO's as equivalent even if they have been produced with slightly different algorithm instances. A string called the key is assigned to each EDO to provide a simple means to distinguish between EDO's of the same type.

EDO type-key

The pair consisting of type and key is called the EDO type-key or simply the type-key. The normal means for retrieving an EDO is to specify event ID and type-key. In any single file, there may be no more than one EDO for a given combination of event ID and type-key.

EDO identifier

We require that objects in one EDO be able to reference those in another, e.g. a track points to its clusters or an electron points to its EM cluster and track. These references must persist even if the two EDO's are in different files and if either or both files or EDO's are replicated. These references are implemented by specifying an identifier for the referenced EDO and an index for the object within the EDO.

Placement category

The ATLAS architecture envisions that the individual EDO is still too fine a level for placing data in files. Instead, files group EDO's into placement categories. These categories hold related EDO's associated with the same event (beam crossing).

The original architecture defines a separate term, sharing categories, for the level at which files can share data. We assume that data is shared with the same grouping that it is placed and do not make use of this term.

A placement category type is a set of EDO type-keys. A placement category instance is the corresponding set of EDO's for a particular event in a particular file. We often abbreviate each of these as placement category and rely on context to distinguish between the two meanings.

Each placement category type is assigned a name and the definition associated with each name is not allowed to change. The algorithms used to produce EDO's of a particular type-key may change with time but a particular placement category type always contains the same set of EDO type-keys. This greatly simplifies the problem of describing datasets formed by merging data from different production environments.

This restriction might be relaxed to allow for limited evolution but we do not envision allowing arbitrary redefinition of the type associated with a name.

The type of a placement category does not guarantee that all the type-keys are present in a particular instance. It may also be desirable to allow a placement category to include any key for a given EDO type (wildcard key).

Stream

When an event is processed, data is read in from a collection of input streams and written to a collection of output streams. These streams are characterized by their types, which specify the collection of placement categories that are included in each event. There is a name associated with each type. The definition of a stream type is not allowed to change.

Later we will associate additional properties with input and output streams. The term stream may be used to refer to a stream type, the type plus the other properties or an instance of either. Context distinguishes between these.

Production environment

Although somewhat outside our scope, we introduce a simple model for tracking data production because we want to record associations between data and production.

A production environment is specified by

1. Release version
2. Version of the global data
3. Supported EDO type-keys
4. Available algorithm instances and their mapping to EDO type-keys

Production thread

A production thread specifies a series of algorithm instances within a production environment. The parameters of these algorithms specify the EDO type-keys used as input and output. The thread also specifies collections of input and output stream types, a selection algorithm for each output stream and placement policy for the output data. The placement policy is used to determine whether data is written by value or reference and to assign an owner for data that is written by value to more than one file.

The algorithms in a production thread transform one collection of EDO type-keys into another. The thread transforms input streams into output streams.

Job

A job is an instance of a production thread. An event ID list is defined and files are assigned to the input streams associated with the thread. The job is a process that carries out event data processing by looping over the events and running the algorithms and writing the output streams specified by the thread.

We assume each production release, production thread and job is assigned a unique ID for any job that is part of sanctioned production.

Production thread ID

Each production thread used in official production is assigned a unique ID.

Store view

The event ID list and input and output streams used to define a job define a view of the event store (or views of the event stores). The store view is the interface between the event-processing framework (Athena/StoreGate in ATLAS) and the persistent event store.

Replicated data

Data may be replicated at two levels. Most trivial is the file level where a file is copied to make it more easily accessible at another site or to change to a more convenient format. The desire to keep file replicas in the same file system as the original file implies some decoupling of file identification and file name.

There is also the possibility of EDO replication where individual EDO's are copied from one or more files to a new file. Replicated EDO's carry the same ID as the original.

Regenerated data

It may be desirable in some cases to regenerate data because the original data is lost or difficult to access. Regeneration is done at the EDO level. The regenerated EDO's are assigned new ID's but also carry the original ID to indicate their origin.

File

An event data file is a computer file on disk, tape or other medium that contains event data. Each EDO written out from transient store is owned and contained by one file. Replicas of the EDO may be contained in other files. A file containing an EDO must be located before that EDO can be loaded (read into memory).

Event data files will be called data files or simply files where there is no danger of ambiguity with other file types. There is a distinction between a physical file that resides in a file system and the logical file that corresponds to that file and all of its replicas. The contents of files are cataloged using logical files. A mapping between logical files and their physical instantiations allow users to locate the actual data.

Each data file is part of a stream or, more precisely, each data file is associated with a particular stream type. Each file contains data for a well-defined collection of event ID's and the file includes one event for each such ID. The data for each event includes all the placement categories defined by the stream type and all or a subset of the EDO type-keys for the placement category type.

A placement category in a file may be replaced by a reference to another file where the placement category may be found. In any file except its owner, an EDO may be replaced with a reference that holds an EDO identifier. There is an important difference between placement category and EDO references: a placement category reference can only be satisfied in the referenced file. An EDO reference can be satisfied by the original version of the EDO or any replica.

Logical file name

Each logical file is assigned a unique name called the logical file name. If a file is replicated, then the new file carries the same logical name. If only a portion of the file is replicated then the new file is assigned a new name. Logical file names within official ATLAS production are unique.

File ID

Each logical file is assigned an identifier called the file ID so that it can be referenced from other files. . There is a one-to-one mapping between file ID's and logical file names.

File format

A file format specifies the physical structure of the files and software system used to read and write these files. The hybrid event store is capable of supporting multiple file formats. We envision that the first of these will be ROOT format.

Dataset

A dataset is a collection of event data in the same stream that covers a well-known run period (e.g. known integrated luminosity). The collection may include all the data from the period or it may include only that data which satisfies some selection criteria (e.g. events with a muon with transverse momentum above

50 GeV/c). A particular event (more precisely beam crossing or event ID) may appear no more than once in a dataset.

Datasets may be merged and in general will include data produced in different production environments. Each event is in the same stream, i.e. has the same collection of placement categories restricted to the same set of EDO type-keys. However, EDO's from different events with the same type-key may have been produced with different algorithm instances. This enables us to form large datasets without reprocessing data.

3. Use cases

Here we introduce some general use cases (i.e. examples). More specific cases are provided in the sections that follow.

Production

We begin with normal data production. This is the most common mode of data access.

The production manager wishes to reconstruct data taken during the last week. First he constructs an input dataset from the raw data for that period. Next he defines a production thread that defines all the algorithms required for full data reconstruction. This is done in the most recent production environment. Finally he uses this thread to define a series of jobs which collectively select and process files from the input dataset. The output files from these jobs comprise the output dataset.

New production

More data is taken and there are some minor improvements in the production code and algorithm parameters. A new production environment is created with a production thread similar to that used in the previous example. The new data is processed with this thread but the old data is not reprocessed because the change was small and the computing resources are not presently available. The datasets from the old and new productions are merged to create a combined dataset.

Selection

A physics group leader wishes to provide her group with a standard sample of events to be used for high-level analysis. The group leader creates a production thread with the appropriate algorithms and registers it in the current production environment. This thread selects events according to some physics criteria and creates summary output data. The combined dataset from the previous example is processed with this thread and the summary data output makes up the desired dataset.

Simple analysis

A physicist wishes to calculate a cross section or limit based on the group sample plus additional cuts on the summary data. He creates a production thread with a selection algorithm that takes summary data as input and writes an ntuple as output. He applies this thread to the group sample and then processes the ntuple to calculate the quantities of interest.

Analysis with reprocessing

Another physicist wishes to do a similar analysis but using a different electron-fitting algorithm before applying the final selection algorithm. She creates a production thread with the appropriate algorithms that takes the group summary data and the original combined dataset as input. The former is used to define the event loop. References from the summary data are followed back into the original dataset to access the electrons and their constituents. New electrons are created and written to a dedicated stream. New summary data is recorded in a separate stream. The simple analysis from the previous section is then applied to this summary data.

4. Defining the problem

We consider the problem of managing event data. Events begin as raw data and then grow as reconstruction proceeds. There is always the possibility to return to a reconstructed event and add data by carrying out additional processing or redoing some or all of the previous reconstruction. The processing of an event may be carried out over many geographical locations and over a long period of time. We can never say that the processing of an event has been completed.

The data for each event is a collection of EDO's. The reconstruction proceeds by applying a series of algorithms each of which takes one or more EDO's as input and produces typically one EDO as output. Our problem is one of object (EDO) management.

There are commercial object databases (ODB) such as Objectivity and the upcoming version of Oracle that handle such data. Licensing costs are high but developing an equivalent solution will also be costly. If the ODB is given full object management, i.e. it manages the files containing the objects and the associations between objects, then it will be difficult to change management at a later date. It is also difficult to integrate other file formats.

The event nature of our problem simplifies matters somewhat. We do not need to support arbitrary associations between objects. EDO's only need to reference other EDO's in the same event and it is unlikely that we need direct bi-directional associations. Also, for the bulk of our production, all or a large subset of events are processed in the same way in a serial manner. In this case it is natural to carry out and track production in terms of files rather than objects or events.

Here we describe a model that explicitly deals with both EDO's and files. Bulk production is managed in terms of files but users are provided with the ability to find the data (the EDO's) associated with a particular event. The scope of this search can be as small as a single file or as large as the complete set of registered ATLAS data. Users will typically deal with standard file collections known as datasets.

We identify the following components in the model:

- File location catalog (5)
- File content catalogs (6)
- EDO catalogs (7)
- Datasets (8)
- Data references (9)
- Data sharing (10)
- Athena/StoreGate (11)
- Generic hybrid event store (12)
- Athena/StoreGate and the hybrid event store (13)
- HES file interface (14)
- File format and streaming objects to and from files (15)

We consider each of these separately in the following chapters (indicated above in parentheses).

5. File location catalog

A file location catalog, also known as a replica catalog, allows users to discover the physical location of a file using a logical file name or identifier. We assume a one-to-one mapping between file identifier and logical file names and speak only of the former in the following.

Use cases

Local file system

A user wishes to determine if a particular logical file is present on a local file system. The user consults the location catalog for the local file system to see if it is present and to obtain its physical location.

ATLAS

A user wishes to determine the nearest location of a file. He consults the global ATLAS file location catalog to find the locations of all registered versions of the file (original and replicas) and then selects the entry corresponding to the nearest location.

Design

File location catalog

If a physical file is specified by site, directory and name, then a file location catalog might be implemented as database table with four columns.

Logical file name
Site
Directory
Filename

This table enables one to find a physical location of a file from its logical name.

The actual implementation will not be so simple and will need to be implemented in the grid environment. We plan to make use of grid replica catalog tools as they become available.

6. File content catalogs

An event data file contains data for a set of events (beam crossings). Each event data file is associated with a stream that defines which placement categories are present in the file and each placement category defines the allowed EDO type-keys. File content catalogs enable users to determine which logical files hold data of a particular type (stream, placement category or EDO type-key), over a specified event range and for a given production thread.

It is expected that most physicists will deal with datasets and will not need to access the file content catalogs. These catalogs will be most useful to those tracking data production and those constructing datasets.

Use cases

Placement category

A user wishes to find all the files containing data for a particular placement category and production environment.

EDO type-key

A user wishes to find all the data for a particular EDO type-key produced with a specified production thread.

Event selection

Either of the above users wishes to restrict the search to a specified even ID or range thereof.

Requirements

1. A means is provided to select files from official production using the following criteria:
    a. Stream
    b. Placement category
    c. EDO type-key
    d. Algorithm characteristics
    e. Job
    f. Production environment
    g. Production thread
    h. Event ID
2. Files are specified by logical name or ID.

Design

File content catalog

The file content catalog can be implemented as follows:

Logical file name
File ID
Stream type name
Min event ID
Max event ID
Job ID
Production thread ID
Production environment ID

There is one entry for each file. Logical files can be selected on the basis of stream type, event ID, job ID, and production thread and environment. The event ID limits can be used to eliminate most files that do not hold a particular event ID. The remaining files have to be checked individually to determine which actually hold data for the specified event.

The job and production information correspond to the job that produced the file. If an EDO in the file is a replica, then this may be different than the job used to produce the original EDO.

Stream catalog

The above table does not allow direct selection based on placement category. In order to select on this basis, the user must first find the streams containing the desired placement category and then make a selection using those streams. The relevant streams may be found in the stream catalog that might be structured as follows:

Stream type name
Placement category name

There is an entry for each placement category in each stream. Production thread and environment ID's are absent because the definition of the placement category does not change.

Placement catalog

If the user wishes to make a selection based on EDO type and/or key, then he must find the placement categories holding the types and keys of interest. This is accomplished using the placement catalog, which can be structured as follows:

Placement category name
EDO type
EDO key

This table has an entry for each EDO type-key in each placement category.

Note that, in general, it is not guaranteed that every or any event in a file will contain the placement category.

Stream and placement category evolution

It is likely that ATLAS will eventually want to modify the definitions of its standard stream and placement category types. Including versions in the names and using aliases, e.g. AOD-3.0 as the stream name with an alias of AOD, can support this. It might be useful to include the aliases in the file content, stream and placement catalogs. Another option would be to allow placement category types to grow, i.e. to later add a type-key to an existing placement category.

7. EDO catalog

At the lowest level, one might want to locate EDO's specified by EDO ID or by event ID and type-key. This could be at the level of a file, a dataset, a site or the ATLAS grid. The file content catalog enables us to locate files that may contain data for the event ID of interest. This combined with the requirement that individual files are required to return information about their contents enables us to access such information at the price of locating and reading physical data files. If there are files for which this information is accessed regularly, it may be worthwhile to record it separate from the data. This is the role of the EDO catalog. It could also be used to catalog data within a file.

Use cases

Event ID

A user wishes to find whether a particular event ID is represented in a dataset or wishes to build a list of the event ID's in a dataset. He consults the EDO catalog for the dataset.

Replicated EDO

A user has an EDO reference and wishes to locate a nearby file containing a replica of the referenced EDO. None of the original file or its replicas is available locally. The search might be allowed to encompass regenerated data.

Design

The EDO catalog might be structured as follows:

EDO ID
Original EDO ID
EDO type
EDO key
Event ID
Placement category name
File ID

There is one entry for each EDO in each cataloged file. The original EDO ID differs from the EDO ID only if the data has been regenerated. The placement category may help to locate the EDO in the file.

The file ID is required even if the EDO ID carries a file ID because the latter is the ID of the original file containing the EDO and will differ from the former if the EDO has been replicated.

We could add parent EDO ID's to this table at a significant increase in size.

8. Datasets

The ATLAS experiment will collect a vast amount of data and no advances in computing are expected to enable individual members of the collaboration to directly manipulate more than a tiny fraction of data. Instead individuals identify criteria for selecting events of interest, physics groups combine these requests to identify shared criteria and then data samples are selected in accordance with these criteria. Moderate-sized samples may be processed in a common way for shared use within a physics group and small samples may be analyzed or further processed by individuals.

This data reduction takes place in a serial manner with the possibility of further production (creation of new data) or selection (identification of events of interest) at each step. It is also possible to merge datasets which are consistent (same stream) and do not overlap (share events). The first step is data acquisition where a series of hardware and software triggers provide the selection criteria for deciding the beam crossings for which data are recorded. This is the only irreversible decision in the sense that we can never go back and recover the data for the crossings that were no recorded. However there is considerable expense associated with processing a large fraction of data and so offline production or selection that traverses such samples cannot be repeated many times.

A dataset is defined as a collection of data that have been acquired over a well-known run period and have been processed and filtered consistently. These samples are the basis of physics analyses that result in discovery, setting limits or measuring cross sections, widths and other physical observables. Most of these depend on understanding the conditions of the accelerator and detector during the run period and on the selections being applied in an unbiased manner.

We identify the following characteristics of a dataset:

1. Well-known conditions of period of data acquisition (especially integrated luminosity)
2. Processing done on the data
3. Selection criteria
4. Corresponding event ID list

5. The type of data recorded (e.g. EDO types and keys)
6. The data

It is useful to take a serial view where a new dataset is formed by processing or filtering the data in an existing dataset or by combining existing (consistent and nonoverlapping) datasets. The starting point is raw data acquired from the detector over a specified run period.

A dataset is very similar to the (event) collection that is described in the ATLAS architecture document and to the event collection familiar to ATLAS objectivity users.

Use cases

Raw data

Data is collected over a three-month period. Files containing good data are identified and entered into a file list and the integrated luminosity is calculated. These quantities are used to create a raw dataset.

Production

The previous raw dataset is processed using the full reconstruction thread from the current production environment. The output is the default reconstructed dataset.

MET selection

A physicist searching for new physics wants to find which events in the processed dataset have missing transverse energy greater than 200 GeV. She creates a thread to do this selection and uses it to process the default reconstructed dataset. The output is summary data and an event list. These comprise the MET dataset.

Reprocessing

Another physicist wants to refit the muons used to construct the MET dataset. He constructs an appropriate production thread and then uses the MET event list and default reconstructed data as input. He fetches the event list from the MET dataset and the data from the MET dataset and processes those events to refit the muons and reconstruct the summary data. The output is two datasets: one with the reprocessed muons (and possibly references to the other EDO's in the default reconstructed dataset) and the other with summary data.

Design

The datasets could be specified in a table as follows:

Dataset ID
Parent dataset ID
Second parent dataset ID
Production thread ID
Name of the stream type
Event ID list
File list
Excess event flag
Integrated luminosity

The parent ID indicates the dataset from which the new dataset was formed. A second parent ID allows the possibility of merging datasets. The production thread ID indicates the thread used to process and select the data. The ID's for the selected events may be found in the event ID list and the logical names for the files containing the selected data are in the file list. One of these may be omitted.

The excess event flag is set true if the file list includes data for events outside the event ID list. This makes it possible to refine an event selection without making a physical copy of the event data.

The table also includes a column for integrated luminosity.

9. Data references

The hybrid event store provides many opportunities to hold data by reference:

1. An event in a file is made up of a collection of placement categories. Each category may be held by value or by reference.
2. The data associated with a placement category in an event is made up of a collection of EDO's each of which may be held by value or reference.
3. An event data file maintains lists of the parent and referenced EDO's for each EDO in the file. These lists are EDO references.
4. An object in one EDO can reference an object in another EDO.

We defer the discussion of placement category references to the following section on data sharing. Here we discuss references to EDO's and the objects inside EDO's.

Use cases

EDO reference

A store view is queried for the parents of an EDO. The response is a collection of EDO references that can be used to locate and load the corresponding EDO's provided they are within the scope of the view.

Object reference

An object inside an EDO references an object inside another EDO. This reference is made persistent across files, i.e. if the two EDO's are saved in different files, then the reference can be satisfied in a subsequent job by providing both files as input. The files may have different formats.

File replication

The reference remains completely valid in the subsequent job if either or both of the input files are replaced with replica files.

EDO replication

The referring and referenced EDO's are each replicated in new files with different logical identities. The reference in the subsequent job is satisfied if either or both of these new files are provided in place of the file containing the original EDO.

File knowledge

If an attempt is made to use a reference and the referenced object or its replica are not provided, then it is possible to directly determine the logical identity of the original file containing the object. This enables the user or the system to easily locate this file or its replica.

Requirements

1. References to EDO's are made persistent by recording an integer index that specifies the ID of the referenced EDO.
2. This index must be unique within its context.
3. Given an EDO ID, a store view can locate and load the persistent representation of an EDO if the EDO is within the scope of the view.
4. EDO references can be chased between different files and different file formats within the store view.
5. Any replica (same EDO ID) of an EDO can be used to satisfy an EDO reference.
6. An EDO identifier is able to provide the identifier of the file containing the original version of the EDO. This aids in file location or error reporting when the referenced EDO is not found in transient store.
7. There is a means to obtain the Event ID, type and key of an EDO from its identifier. This makes it possible to determine this information without chasing the EDO.
8. The indices used to represent identifiers may not change when an EDO is replicated. This allows bitwise copying of EDO's without the need to update EDO references.
9. References between objects in (persistent) EDO's are made persistent by expressing them in terms of object identifiers (OID's) that specify the EDO ID and an object index giving the location of the object within the EDO.

Design

Here we are primarily concerned with the nature of the EDO ID and the OID. The HES view is described in a later chapter describing the generic hybrid event store.

EDO and object identifiers

An object identifier (OID) is composed of an EDO ID and an object index. A store view can use the EDO ID to locate the EDO if there is a copy within its scope. The EDO can then use the object index to locate the object within itself. (This indexing is trivial for vector containers but requires a bit of extra work for more complex containers such as maps and multimaps.) The persistent representations of the EDO ID and OID are integer indices.

Counts

We make estimates of the number of bits needed to represent various indices. We assume data will be taken at a rate of 100 Hz for 2x10^7 seconds/year for 20 years, allow an average of 5 MB per event to allow reprocessing and assume the average file size is 1 GB. We assume 1000 EDO/event or an average EDO size of 5 kB. We obtain the following counts:

	Number per year	Number in 20 years	Number of bits (20 years)
Events (beam crossings)	2x10^9	4x10^{10}	36
Files	1x10^7	2x10^8	28
EDO's	2x10^{12}	4x10^{13}	46

The last column gives the number of bits required to represent the 20-year count. We have not included Monte Carlo data but expect it to roughly double these numbers.

File index

Each file or at least each file that is part of official production must is assigned a unique ID represented by an integer index. The table indicates that a 32-bit index with one bit to indicate Monte Carlo has only a three-bit (factor of eight) safety margin. This small margin would require close coordination between all production centers and leaves little or no room to include information about the file format.

Another option is to independently generate indices perhaps by combining machine and process ID's with a time stamp. This would require something like 128 bits.

We assume an intermediate solution with a 64-bit ID. This leaves many extra bits that can be used to simplify coordination and to encode additional information. We allocate eight of these bits extra bits to encode the type of the file format including non-HES formats.

EDO ID and index

Each EDO has a unique ID and is referenced via an index that must be unique within its context. We have required that EDO ID's provide access to the ID of the file owning the EDO, the event ID and the type-key. We begin by defining a complete index where all this information is carried locally, i.e. can be obtained directly from the ID without consulting external maps. This index is fairly large and a scheme for reducing the index size is described below.

The complete file index holds separate indices for the file, event, type and key. The file index is just the 64-bit index described above. We conservatively assume another 64 bits for the event ID. We allow 32 bits for the type identifier. The key is a string and we allow its index to have a variable length to avoid wasting space or imposing a severe length restriction. We use the first byte to hold the length and then store one character in each subsequent byte. We use an integral number of 32-bit words so that the EDO indices can be aligned on 32-bit boundaries. The size of an EDO index is

192 + 32*[nch/4] bits

where nch is the number of characters in the string.

Most EDO references will be made within a context and will hold a much smaller restricted index. The context will provide a table mapping the restricted indices to the complete indices. One such context is transient memory and is provided by the HES EdoId class. ID's are assigned a 32-bit transient index when they are created from a complete index. ID's created from the same complete index are assigned the same transient index so that comparisons (ordering and equality) can be based on the smaller index.

Each EDO read from or written to store defines a restricted context that includes all referenced EDO's. The set is limited to 512 entries so that this restricted EDO index can be expressed in eight bits. The table mapping restricted indices to complete indices is stored along with the EDO.

Object indices

The complete identifier for an object in an EDO is an EDO identifier plus a local object index that specifies the object within the EDO. The corresponding complete object index combines an EDO index and the local object index. There can be many such indices in the persistent data and so it is important they be kept small.

To meet this requirement, we take advantage of the fact that we do not need to support arbitrary references. Each EDO has a small set of EDO's that contain its referenced objects. We call these the referenced EDO's.

We require each EDO maintain an indexed list of its referenced EDO's as described in the preceding section. The full object index is a combination of this restricted EDO index and a local object index. Clearly this restricted EDO index is valid only in the context of the referencing EDO and must be translated into or from a transient index when data is moved from or to persistent store.

The complete object index is recorded as an 8-bit local EDO index and a 24-bit local object index. This allows an EDO to have up to 16 million contained objects and 512 referenced EDO's. The size of the complete object index is 32 bits.

10. Data sharing

ATLAS will acquire and produce vast amounts of data (10 PB/yr) but any individual physicist will only be able to examine a tiny fraction of this data. Physicists will examine certain types of data (EDO type-keys, placement categories and streams) for specified collection of events (datasets). Each analysis will ultimately require its own subset of the data but there will be considerable overlap in both data type and event selection. This overlap makes it possible to share data at many different levels. These are discussed in the following sections.

The motivation for data sharing is to save both CPU cycles and space on disks or other physical media.

Datasets

The highest level of sharing is the dataset. All recorded data will be organized into raw datasets and the processing of these is the first step in data reduction. This processing and thus the input and output datasets are common starting points for all analyses. Event selection is then carried out in a sequential manner. Each step produces a new dataset that is likely shared by a smaller set of analyses.

Exclusive streams

Exclusive streams are another way to share data at the level of a dataset. Datasets are formed with a series of exclusive streams and then these are combined to form higher-level datasets. An example would be to create separate streams (and datasets) for events with one, two and three jets. These could then be combined to form a dataset with one or more jets and another with two or more jets.

Files

Files are both a logical level of data organization and the physical medium by which we record the data. The sharing of logical files is via datasets as described above. Physical files are shared by replication. A physicist at one site can copy the files of interest from another site.

Placement category

The data for an event in a file is organized into placement categories. Two files may share the data in a placement category either by replication or by reference.

EDO

EDO's may be shared between files. Any file may contain the EDO data or an EDO reference. The latter is the same as that used in object references. Unlike the placement category reference, any replica of that EDO or even a regenerated version, not just the version in the referenced file, may satisfy the EDO reference.

Sharing categories

The original architecture document uses the term sharing categories to describe the level at which events share data between files. We avoid this term and provide this functionality at the level of both placement category and EDO.

Use cases

Shared placement category

The initial reconstruction stream includes separate placement categories for tracking and calorimeter data. A dataset is created in this stream. A physicist wishes to select events with high-E_T jets and create a new dataset in which the calorimeter information is replicated but the tracking data is not replicated to save space but is referenced. The new dataset by itself is processed with a new jet algorithm. That output dataset from that processing is used in combination with the original dataset to study the matching of tracks to the new jets.

Overlapping streams

A job is run with two output streams that share a placement category. The two streams have different event selection criteria. Data in that placement category is always written to the first stream. It is written to the second stream only if the event is not represented in the first stream. Otherwise it references the data in the first stream.

EDO

The data in the tracking category from the first example includes clusters, track seeds, initial tracks and refit tracks. The physicist is only interested in the EDO containing refit tracks and references that EDO instead of the full tracking placement category.

Requirements

1. The data for any placement category and event in a file may be replaced with a reference to the same placement category and event in another file.
2. The placement category reference includes the ID of the file containing the data for the category. This enable the system to locate the file or report its identity if it cannot be found.
3. The data for any EDO can be replaced with an EDO reference. The latter is the same as that used for object referencing.

Design

The sharing of files between datasets is provided by the design of the datasets. Placement category and EDO references are handled by the HES view as described below.

11. Athena/StoreGate

We now consider the problem of the user interface to the hybrid event store, i.e. how clients store and retrieve event data. Our primary goal is to provide access within the ATLAS event-processing framework Athena[4], which is based on Gaudi[5]. Athena uses StoreGate[6] to manage transient data and to provide the interface for converting event data between transient and persistent forms.

We would like to minimize physical coupling and make much of our software independent of Athena, StoreGate and other ATLAS systems. We identify a system called HES that provides a generic description of the hybrid event store and a system called AHES that provides the connection to ATLAS. Use cases, requirements and some of the design for these systems are described in the following chapters. Our emphasis in this chapter is on the changes required in Athena/StoreGate.

Use cases

Hybrid event data input

A user wishes to read data from the hybrid event store, specifically to loop over all the events (event ID's) in a dataset. When an event is processed, all data (in the dataset) for that event is accessible from the StoreGate transient data store. A job is configured with a single input stream corresponding to the dataset.

Multiple input streams

The above user can read more data for each event by providing additional input streams. For example, calorimeter data might be read from one stream and tracking data from another. Or summary data might be read from one stream and full reconstruction data from another. One stream is used to supply the event ID and all other streams are searched for matching data.

Restricting the data in each event

A user wishes to read only a subset of the data in a stream. For example, the user might want to read only the calorimeter data from a stream containing reconstructed data for all detectors.

Chasing EDO references

An electron in one EDO references a track in another EDO. A user accesses the electron EDO from the transient store and chases the reference to obtain the EDO holding the track. The latter need not be in part of the input streams.

Hybrid event data output

At the end of processing an event, data in the transient store is written to the hybrid event store. The output is to a collection of streams whose definitions specify which data (i.e. which placement categories and hence which EDO's) are recorded in each stream. There is a selection algorithm associated with each stream that is used to decide on and event-by-event basis whether the stream is filled. There is also a placement policy that is used to decide which placement categories and EDO's are written by reference and which are written by value. If a new EDO is assigned to more than one stream, then the policy also determines which stream is used to assign its identity. The other streams carry replicas of this EDO.

Other store types

A user wishes to read GEANT data from ZEBRA, early reconstructed data from Objectivity and later reconstruction streams from the hybrid event store and then write data to the hybrid store. The transient data for each event includes contributions from all three and the user is able to follow object references from HES to ZEBRA.

Requirements

1. Athena/StoreGate can be run with input from any of the supported persistent store types. To begin with, these will include ZEBRA and the hybrid event store. Any combination of these is supported.
2. Physical dependency between Athena/StoreGate and any of the store types should be minimized. In particular, the headers or libraries for one store type are not needed to build or run an executable that does not use that type of store.
3. Athena provides means to specify and configure the persistent stores that are to be used. The configuration includes a means for assigning an ID for each event, input streams that define the visible part of each input event, and output streams that specify how data is to be written out. The output streams are named and include a selection algorithm that is used to decide whether each event is to be included. Each store may also define a realm of accessible data that is not used to directly define input events but may be used to resolve data references.
4. Athena provides means to read event data from all persistent stores into StoreGate at the beginning of each event. This collection of EDO's is the input data and it does not change during event processing. It is not required that the data be converted (from persistent to transient form) or even loaded (persistent form read into memory) but it is required that StoreGate know exactly which type-keys are present in the event.
5. Athena runs algorithms that may add data to StoreGate. This new data and the input data together comprise the visible data. An algorithm may fetch any visible EDO by requesting it by type-key from StoreGate. The response to this request is a StoreGate data handle.
6. Such a data handle may be dereferenced to gain access to a transient representation of the EDO. The dereferencing is allowed to fail if the object is not accessible.
7. The data handle must distinguish between a type-key that is not present in the visible data and one that is present but is not accessible.

8. No type-key may appear more than once in the visible data. If an attempt is made to add an EDO whose type-key is already represented, then that EDO is rejected and an error is reported.
9. An object in one transient EDO may reference an object in another EDO. The transient representation of this reference is a StoreGate data link. A data link may be dereferenced to gain access to the referenced object if the object's EDO is accessible. This may require loading and converting that EDO. It does not become part of the visible data.
10. These references may point between different store views and store types. This implies output is not done for any store until the new EDO ID assignments are made for all stores.

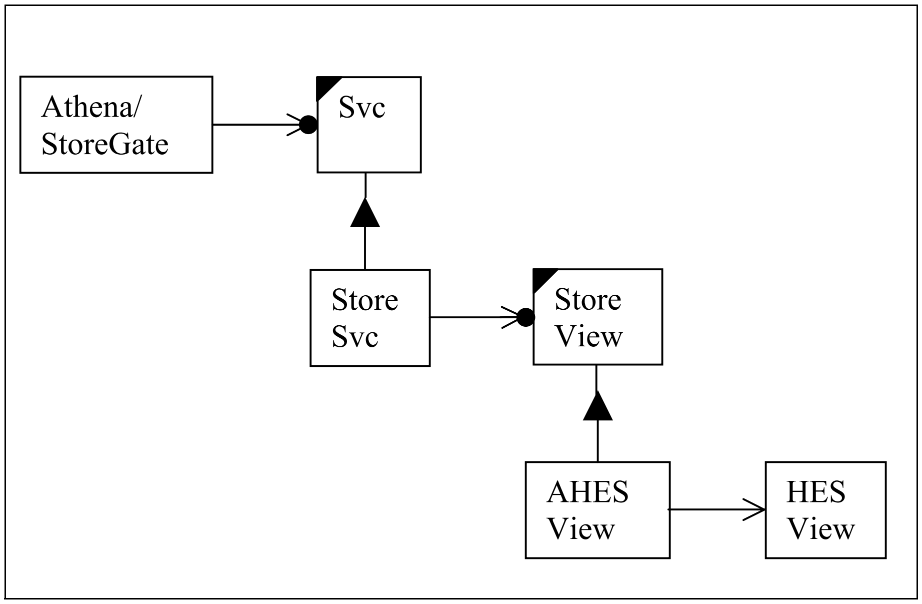
Design

Store service and store views

We begin by introducing the notion of a "persistent event store view" which we will usually abbreviate as "store view". As the name implies, each of these provides a particular view of a particular persistent event store. The view limits the event data which can be read, dictates how to place data that is written and provides a means to iterate over the events (more precisely event ID's).

We envision modifying Athena so that input and output are done through a persistent store service that holds one or more store views. The store service might be divided into separate input and output services.

Direct dependence between Athena/StoreGate and any of the persistent store types is avoided by holding a store view (or views) via an abstract base class. Different store types such as ZEBRA, Objectivity and the hybrid event store are described by different subclasses. This is illustrated in the following class diagram.



Interface between Athena/StoreGate and the hybrid event store.

The choice and configuration of store view subclasses is specified in the usual job options file. Each store view subclass interprets the configuration data for its type.

The most useful case is a single hybrid event store view but we allow for the possibility of processing old data by using ZEBRA or Objectivity views and for simultaneously accessing more than one store type by registering multiple store view in the same job.

Creating store views

We require that Athena and its store service have no dependency on the concrete store view subclasses. This is accomplished by following the Gaudi pattern and making the store views Gaudi services. This enables us to configure the views from the usual Gaudi job options input.

The store service assigns an integer ID called the store ID to each store view. A null value is used to refer to new data, i.e. that produced by an algorithm and not read in from any store. The store view is informed of its ID so it can use it to take ownership of proxies during output (more on this later).

If we assign an index for each store type and restrict ourselves to a single store view for each store type, we can use this index as the store ID.

Transient event store

The StoreGate transient event store includes a pool of transient data objects and a pool of proxies that are cleared and rebuilt every event. The former includes all EDO's that have either been newly created by an algorithm or loaded from a persistent store and converted from persistent to transient representation. There is a proxy for each of these transient data objects and, in addition, there may be proxies for other EDO's that have not yet been converted or loaded.

The transient store includes both visible data and reference data. Visible data is that which is part of the input event or was added by an algorithm. Algorithms fetch data by type-key and only visible data is directly accessible in this manner. Each type-key can only appear at most once in the visible data. If an attempt is made to add an EDO whose type-key is already represented, the new EDO is rejected and an error is reported. Any data added by an algorithm is visible.

Objects inside one EDO may reference those in another and it may be possible to gain access to the EDO holding the referenced object even if that EDO is not part of the visible data. In this case, that EDO becomes part of what is called reference data. It is not directly accessible with a type-key reference and is not available for output. It is stored in the reference part of the transient store so that it is readily available the next time it is needed to satisfy an object reference.

EDO ID's

We have maintained the requirement that a type and key uniquely specify an object in the visible part of the transient data store. However, we recognize that the persistent store may contain other versions of that EDO in the same event with the same type and key. A reference to one of these other versions should not be satisfied with the version that happens to be in the transient store.

We avoid such ambiguities by adopting the HES concept of an EDO ID, i.e. a unique identifier that labels each EDO. For persistent store types such as HES that provide such an ID, we adopt their ID's possibly with a translation to a StoreGate-defined type. For persistent store types such as ZEBRA that do not provide an EDO ID, we assign one.

Proxies

A proxy for an object read in from a persistent store may be in one of four states:

1. Data not loaded; proxy carries the type, key, store ID and EDO ID.
2. Data loaded but not converted; proxy carries the above plus the file format (if the store type supports multiple formats) and the IOpaqueAddress of the EDO.
3. Data retrieved and converted; proxy carries the above plus the address of the transient representation of the EDO.
4. Data found to be inaccessible; proxy carries the same as 1 plus a flag is set to indicate the inaccessibility.

At present the Gaudi IOpaqueAddress includes the type, the key, the Gaudi StorageType and the address of the persistent representation of the object. This will be interpreted or expanded for HES to hold a reference to the file and the location of the object within that file.

A proxy may be asked to return a pointer to the transient data object. If its data has not been loaded (the IOpaqueAddress is null) and the proxy is not marked inaccessible, the proxy updates itself by passing the store ID and itself to the store service. The service passes the proxy to the appropriate store view, which attempts to locate the EDO and load its persistent representation. The view has the option to update other known proxies, e.g. those in the same placement category.

Control is then returned to the proxy. If it is now marked inaccessible, null is returned. Otherwise the IOpaqueAddress will be set. If the data is not converted (the transient address is null), the proxy calls the appropriate converter with the IOpaqueAddress and receives and sets the address of the converted object. This address is returned.

The proxy for a new EDO, i.e. one added by an algorithm, will be in a different state. It holds the type, key and transient address. The store and EDO ID's may be added later after a store view takes ownership and assigns an EDO ID.

Data links

References to objects inside of EDO's are expressed with data links. The data link holds an index indicating the location of the object inside an EDO. When the data link is fully updated it also holds a proxy for the referenced EDO. When dereferenced, the data link updates itself, fetches the transient address of the EDO from the proxy, uses the location index to find the address of the referenced object inside the EDO and returns that address. A null address is returned if the proxy cannot be found or if the proxy is marked inaccessible.

For the purpose of accessing data in the transient store, a data link may be created in any of three states:

1. with type and key
2. with type, key and EDO ID
3. with type, key, EDO ID and store ID

A new data link (one not read in from persistent store) may be created in any one of these states. Data links read in from persistent store are in the third state. The type, key and EDO ID are part of the persistent data and the store ID is filled in when the object is loaded or converted.

In the first state, the data link is updated by asking StoreGate for a proxy with a matching type-key in the visible data. In the second state, the search includes both visible and hidden data and additionally requires that the EDO ID match. In both cases, the proxy for the EDO is assumed to already be present in StoreGate. In the third state, StoreGate will ask the specified store view to try to locate the EDO if it is not already present in the transient store.

When a data link is converted to its persistent representation, its EDO ID is taken from its proxy if the ID is not yet assigned. There is an error if the proxy does not hold an EDO ID at this time.

It is possible that the persistent representation of a data link will require additional information such as event ID, reference EDO ID or store ID for conversion to transient form. This data might live in a history object associated with the EDO.

Users may also record references by creating a data link from a bare pointer to an EDO. StoreGate must provide the means to locate the proxy associated with that object so the data link can be written to persistent store.

EDO ownership

Each input EDO is owned by the store view that read it in and may only be written out by that same store view. EDO's created by algorithms initially have no owner. At the end of processing an event, each store view is given the opportunity to take ownership of any EDO's that have not yet been claimed. Each of these EDO's may only be written by the store view that has taken ownership.

Ownership is established by assigning the appropriate store ID to the proxy for the EDO. Store views are also expected to assign EDO ID's when they take ownership. These are recorded in referencing data links and aid in navigating back to the referenced EDO after the data is written out to persistent store(s) and later read back into transient store.

Assigning the event ID

The store service is called at the beginning of each event to assign the ID for that event. It normally does this by asking one of its stores to return the ID for the next event. A job configuration parameter indicates which store has this responsibility. The service may be configured to take the event ID from another source, e.g. through interactive user input.

Reading the input event

The store service defines the input event by looping over all store views and passing each the event ID. Each view inserts a proxy in the visible part of StoreGate for every EDO it defines as part of the input event. This is the only opportunity that a store view gets to add visible proxies for the event. After this loop over store views, the input event is completely defined, i.e. all proxies are present in StoreGate. This does not imply that any input EDO has been converted, loaded or is even accessible. The use of proxies allows us to defer these operations until a user attempts to access the data in the EDO.

Retrieving data

Algorithms retrieve visible data from StoreGate by providing a type-key and receiving a data handle that points to the corresponding proxy. The proxy is missing if the type-key is not found in the transient data store. When the data handle is dereferenced, it asks its proxy for a pointer to the transient representation of the EDO. If needed, the proxy will locate, load and convert the EDO as described above.

Object references

An object in one EDO may reference an object in another EDO. These references are expressed in terms of the data links described above. Data links may be dereferenced to access the referenced object.

Storing data

Event data is written out to one or more output streams. The output store service associates a name, a store view and a selection algorithm with each stream. Each store view may associate other data (such as placement information) with its streams. The information for configuring the streams is taken from the job options input.

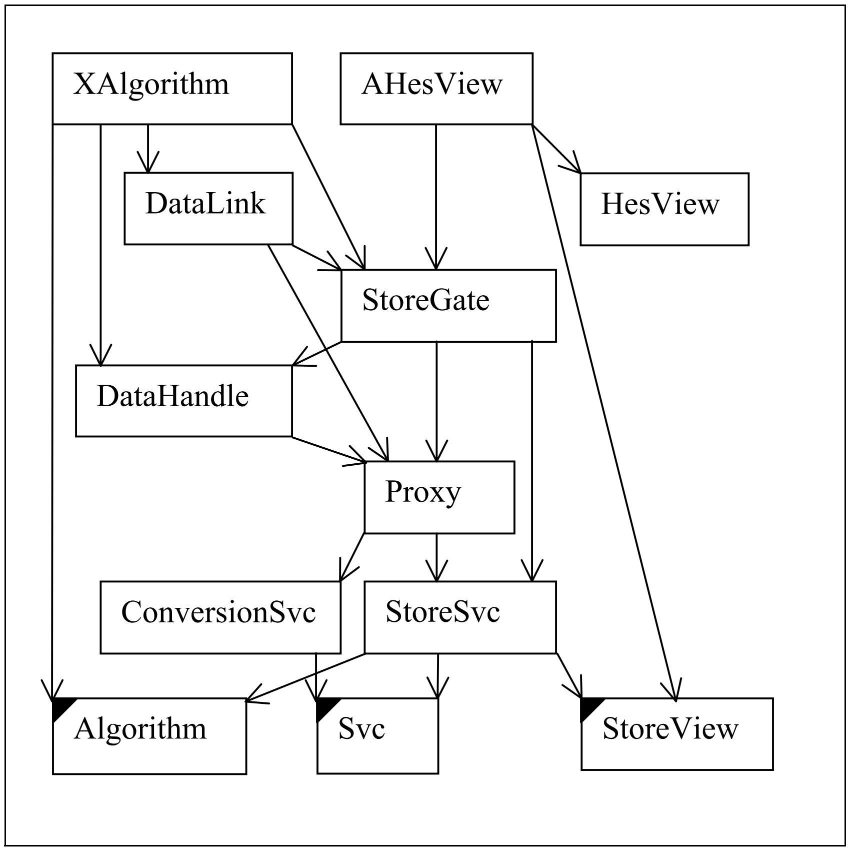
At the end of processing for an event, the transient store includes input EDO's which are owned by store views and new EDO's that do not have owners. The store service applies the selection algorithm for each output stream and builds a list of accepted streams for each store view. The list is passed to each view, which can take ownership of any of EDO that is not yet owned. This is done by assigning store and EDO ID's to the EDO's proxy. Any EDO's that are not claimed are not written out.

After these assignments are made for all views, the service again loops over store views and asks each to write the data for its streams.

It is assumed that any owned EDO was read in or will be written out (by value) by the owning store. It is further assumed that the each owned EDO has been assigned a meaningful EDO ID (through its proxy). If these conditions are met, then data links referencing any owned EDO can be reconstructed after being written to a persistent store and later read back into transient store.

Physical dependencies

We require that there be no cyclic dependencies in the model. The following diagram shows this is the case.



Athena/StoreGate and HES physical dependencies.

12. Generic hybrid event store

To minimize physical coupling and to allow application outside of ATLAS (including Athena/StoreGate), we would like to put as much functionality as possible into a generic description of the hybrid event store. This system is called HES and, by definition, has no dependence on Athena, StoreGate or any other ATLAS-specific software.

HES has the responsibility of moving persistent representations of EDO's from memory to files in the event store and later moving them back from persistent store into memory. The structure of these objects is defined by their file format. HES is capable of simultaneously managing files in different formats as long as all satisfy a common file interface.

Applications are expected to read and write EDO's in memory. They may restrict themselves to a single file format or may define a common transient representation for each type of EDO. The latter is the choice made by ATLAS. HES takes no responsibility for converting persistent objects to or from their transient representations. StoreGate provides this functionality in ATLAS.

HES provides data access though the HES view which has an interface similar to that of the Athena/StoreGate abstract store view. In ATLAS, AHES view inherits from this interface and makes use of the HES view to access the persistent data. This application is the origin of the use cases that follow but other applications are expected to have similar usage.

Use cases

Single file input

A HES view is configured to use a single file as input. At the beginning of each event, the view finds the next event in the file and uses it to set the event ID and determine the EDO's in the event. Users can query the view for the event ID, the list of EDO's or the persistent address of any of these EDO's. All EDO queries are by type and key. The view may defer location and loading of an EDO until its address is requested.

Multiple file input

A HES view is configured with multiple streams. Each stream is associated with a collection of files with a common stream type and at most one occurrence of any event ID. One stream is designated as the event ID source. The event ID's in that stream are subset of those in any other stream. At the beginning of each event, the view finds the next event ID in the event ID stream and then uses all the EDO's associated with this event ID from each stream to define the input event. The event ID and loaded EDO's are available as before.

Restricted set of placement categories

In the previous case a list of placement categories is provided for one of the streams. The data visible for that stream is restricted to EDO's in those categories.

Placement category references

While reading an event in a file, a HES view encounters a placement category reference. The reference is the logical name of the file that holds the category by value. It searches for the referenced file in its input streams and then in its collection of reference files. When the file is located, the placement category is read from there.

EDO references

A requested EDO is held by reference. This reference is an EDO ID. The input stream and reference files are searched for an EDO with a matching ID.

New data

At the end of an event, a HES view is given a collection of type-keys corresponding to new EDO's that have been produced and are available for output. No type-key may appear more than once in the union of the input and new data.

Assigning EDO's to output streams

A HES view is configured with any number of output streams each of which has a stream type. At the end of event processing, the HES view is given the list of new EDO's (by type-key) and the list of accepted streams (by name). The HES view assigns subsets of its input EDO's and these new EDO's to placement categories in its output streams. Each is assigned to be written either by value or reference. The HES view assigns an ID to any new EDO's that are to be written. The view can be queried for any of these assignments and for the file and file format associated with any stream.

Writing output streams

The user (e.g. the AHES view) converts the newly assigned EDO's to their persistent representation(s) and calls the HES view to write the data for the event.

Requirements

1. HES provides a means to access placement category and stream type definitions.

2. A HES view is constructed and configured at the beginning of each job and provides the interface to HES. The view takes care of file management.
3. A HES view has a collection of input stream definitions that are used to define the input events. Each stream is associated with a collection of files. The stream has type and each file is associated with that same stream type.
4. The visible part of any input stream may be limited to a subset of its placement categories.
5. One of the input streams may be tagged as the event ID source.
6. A HES view has a second collection of files that are used in addition to those in the input streams to chase placement category and EDO references. These are the reference files.
7. A HES view defines the output streams. Each stream has a name, a type, a file format and policies for naming, opening and closing files. The view also has a placement policy that determines whether EDO's are written by value or reference. This policy also assigns ownership to one file when an EDO when it is written by value to more than one stream.
8. A HES view may be queried for the next event. This is the next event ID in its event ID source stream.
9. An event is initialized by specifying the event ID. After event initialization, the HES view holds a list of type-keys that specify the visible input EDO's for that event. It holds the ID for each of these EDO's and holds the file ID for any EDO's that have been located (found by value).
10. The input type-keys for an event are the concatenation of those that appear in each visible placement category in each input stream for that event. The visibility of each placement category in each input stream is part of the configuration of the streams.
11. The contents of a placement category are not fully specified by its type and so placement category references must be chased during event initialization to complete the definition of the input event.
12. Any type-key may appear only once in the input event. If different placement categories point to EDO's with the same type-key and different EDO ID's, there is an error.
13. A visible EDO may be located by type-key, i.e. have its EDO references chased until a file containing the EDO by value is located. The search may extend into visible and hidden placement categories in the input streams and into the reference files.
14. Any EDO may be located by EDO ID. (The type and key are specified by the EDO ID.) The HES view first checks for a match in the visible EDO's. If this fails, the hidden placement categories in the input streams and the reference files are searched until a match is found. This is called a reference EDO. A reference EDO is not visible (accessible by type-key) and it may have the same type-key as an existing visible or reference EDO.
15. A visible or reference EDO which has been successfully located may be loaded, i.e. copied into memory with the corresponding address recorded in the view.
16. Any attempt to locate or load an EDO may fail because the data for the EDO is not present in the HES view. In this case the type-key or EDO ID is marked inaccessible.
17. All operations to locate or load EDO's have the option to locate or load any other EDO's. The list of visible type-keys does not change.
18. The HES view can be queried for the list of visible type-keys. For any visible EDO specified by type-key or any EDO specified by ID, the view can be queried for the status (located, loaded or inaccessible), for the EDO ID, for the file ID if located, and for the address of its persistent representation if loaded.
19. The HES view provides an end-of-event-processing method in which it is passed a list of accepted stream names and a list of type-keys for new EDO's. These and the visible EDO's are available for output. There is an error if a type-key appears more than once in the concatenation of these two lists.

20. During end-of-event processing, the view assigns EDO's to output streams either by value or reference, assigns the file for each stream, opens the event in each file and makes an ID assignment for each new EDO that is to be written.
21. When assigning a file for a stream, the view checks if there is room in the presently assigned file and, if not, closes it and opens another.
22. After end-of-event-processing is called, the HES view can be queried (by type-key) for the ID assigned to any new EDO. It can also be queried for the file ID and the type-keys of the new EDO's associated with each accepted stream. The file format for any output stream is always available.
23. The HES view provides a write-event method that is to be called after the new EDO's have been converted to their persistent representations. For each stream, the view writes the data and closes the event. There is an error if any assigned EDO's are unwritten.
24. The HES view may choose to write each placement category by reference or value. Each written placement category must appear by value in at least one input or output file.
25. The HES view may write EDO's by reference or value. Each written EDO must appear by value in at least one input or output file. If an EDO appeared in an input file, then it retains its old ID. If it is a new EDO, then one of the files holding the EDO by value is assigned ownership and is used to assign an ID for the EDO.

Design

The above requirements might be implemented with a pool of proxies similar to those used in the transient world. They would include states to indicate whether the EDO had been located or loaded. The pool would have one entry for each type-key in the input event. A separate pool would hold EDO's which were located when following references.

13. Athena/StoreGate and the hybrid event store

Next we consider AHES, the interface between Athena/StoreGate and the generic hybrid event store, HES. AHES provides the concrete AHES view which is an implementation of the abstract store view used by the Athena/StoreGate store service.

Use cases

1. The AHES view is called by the Athena/StoreGate store service and makes use of the HES view. The construction of the AHES view is triggered by the job options input.
2. The AHES view is configured to chase references into a non-HES store such as ZEBRA or Objectivity.

Requirements

1. AHES provides a means to construct and register an AHES view in the Athena/StoreGate store service. The AHES view configures the HES view using relevant data from the job options.
2. The AHES view implements the store service interface functionality making use of the HES view.
3. Other store views can be registered with the AHES view for use in chasing EDO references.

Design

Initialization

The AHES view is a Gaudi service and its configuration is specified in the job options. The configuration associates input streams with the AHES view. These definitions associate files with each stream in a manner consistent with that required by HES.

The visible part of any stream may be restricted by providing a list of visible placement categories. If the list is omitted, all categories are visible.

One input stream may be designated as the event ID stream. If so, it will be read in a serial manner.

The configuration also defines the output streams and assigns each a name and a selection algorithm. The HES placement policy is also defined.

Event iteration

When an AHES view is asked for the next event, it passes this request to the HES view and echoes the result, translating it if HES and Athena/StoreGate have different event ID types.

Reading the input event

When an AHES view is asked to read an event specified by ID, it passes this request to the HES view and then creates a StoreGate proxy for each visible type-key in HES view. These proxies are added to StoreGate.

Fetching data

The AHES view may receive a proxy in a request to load an EDO. If the proxy has an EDO ID, then that ID is used to ask the HES view to load the EDO. Otherwise the type-key is used for this same purpose. The HES view is then queried to check if the EDO is loaded and, if so, its StoreGate proxy is updated with the persistent address of the EDO. If not, the proxy is marked to indicate that the data is inaccessible.

If other views are registered with the AHES view, the latter will ask those views to try to update the proxy before giving up and marking it inaccessible.

Storing data

After an event is processed, the AHES view receives a list of accepted streams. The view fetches the list of visible proxies from StoreGate and selects those that have not been assigned a store ID. It builds a list of type-keys from these and passes it and the list of accepted streams to the HES view, invoking the end-of-event-processing. The HES view assigns EDO ID's to a subset of these type-keys. The AHES view queries the HES view for these new EDO ID assignments and adds them and the store ID to the corresponding StoreGate proxies.

After all stores have been give the opportunity to take ownership of available EDO's, the AHES view is called to write the data to its output streams. The view loops over accepted output streams and queries the HES view for the file format and the list of EDO's that need to be converted for that stream. The AHES view then converts those EDO's to their persistent representations. Finally, the AHES view calls the HES view and invokes event-writing.

14. HES file interface

HES is able to handle files with different formats and so we require that all HES-capable formats implement a common interface. Here we generate a design for that interface.

Use cases

Association with stream

The library for any active file format is identified at run time. It is linked and registers itself with HES. This registration associates a name with a file creator for the format

HES view

The HES view uses fetches event ID's and EDO's from files and writes EDO's to files. This access is all done through the HES file interface.

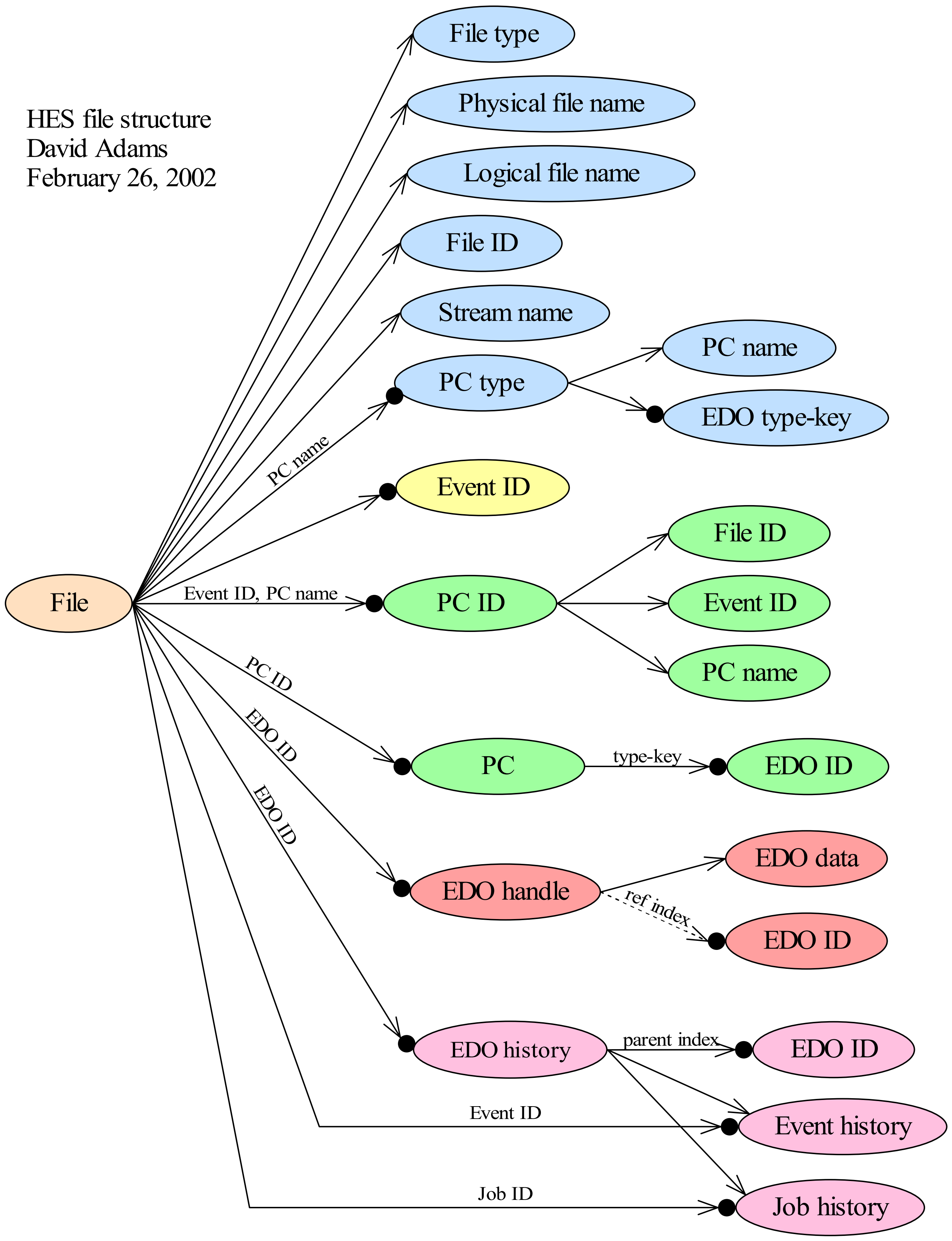
Requirements

1. Files are created in an unlocked state and event data can be added until a file is locked. After locking, the cannot be modified and is guaranteed to be readable. The lock status of a file can be queried. The lock status does not change when a file is closed and reopened.
2. Each linked file format registers a file creator with HES. The file creator can be used to create new files or open unlocked files for writing. It may open existing locked files for reading.
3. Each file is associated with a stream type and carries the list of placement categories associated with that stream type and the list of type-keys associated with each placement category. These definitions may b e queried.
4. Each file is assigned a unique logical name and integer ID upon creation and may be queried for either.
5. Each file holds data for a well-defined collection of event ID's and may be queried for this collection, the number of ID's and the minimum and maximum values.
6. A file may be queried for its size in bytes.
7. A locked file is able to iterate over its list of event ID's, returning a new value upon each request for a new event until the list is exhausted. The ID's need not be returned in order.
8. A file holds the same set of placement categories for every event. (An event is the data associated with an event ID.) No two events have the same event ID.
9. A placement category for any event may be held by reference or value. If by reference, the file holds the name of the referenced file, which holds the placement category by value. If by value, the placement category holds a list of type-keys and an EDO for each. The type-keys are a subset of those that the file associates with the placement category. A locked file may be queried for the information about a placement category by event ID and placement category name.
10. An EDO may be held by reference or value. If held by reference, the file holds the ID of the EDO. If by value, the file holds the EDO ID and the actual data for the EDO and provides a format-specific means to access this data. A locked file may be queried for the value/reference status and ID of an EDO by event ID, placement category name and type-key.
11. An unlocked output file that is not yet open for an event may be given an event ID to open a new event. The file creates all the placement categories in the event. By default, these are by value. The event ID must be on that is not yet included in the file.
12. Event data may only be added to a file with an open event. This data may be a reference to a placement category, a reference to an EDO or an EDO by value. The data is added to the current open event. No new event may be opened until this event is closed.
13. A placement category may be changed to a reference by providing the logical name of a file that holds or will hold (by value) the referenced placement category. The existing placement category must not already be a reference and must not hold any EDO's.

14. An EDO reference is added to a placement category by providing the name of the placement category and the type, key and ID of the EDO. The placement category must be one that is written by value and does not yet include the specified type-key.
15. An EDO reference may be changed to an EDO value by providing the address of the persistent representation of the EDO and the type-key.
16. The event may be closed any time after opening. The event may not be reopened.

Design

The above interface is implemented by defining an abstract class HesFile with corresponding methods. There is also a file manager HesFileMgr where file formats register their creators and the HesView goes to locate files.



The following figure shows the implied structure of a HES file.

15. File format and streaming

We need to support random access, i.e. fetch a particular EDO from a file without reading through the entire file. On the other hand, the bulk of our processing will likely be in a serial mode so we do not want to sacrifice performance in this mode to enable the former. Both of these assume normal data access through converters where data is input and output via the StoreGate transient data store. Data browsers used for analysis may be direct, i.e. not use conversions, and hence require that the data be in their native format. An obvious example is ROOT.

The following considerations enter into the choice of the file format and means of streaming:

1. Input and output conversion
2. Speed of access in serial and random access modes
3. Management of object references within a file
4. Direct access browsing (no conversion)
5. HES interface

Input and output conversion

A file holds a collection of objects including EDO's and histories. Each object has been converted from its form in memory to its persistent representation and will be converted back when the file is read.

Atlas has an active program for defining a data definition language (DDL) that specifies a language called ADL for defining data object classes. It automatically generates converters for both Objectivity and will do so for the BNL ROOT format. There are efforts to develop another DDL in the Gaudi framework and a similar effort in ROOT. We assume ATLAS will settle on one definition language for which we use the name ADL.

We are left with the requirement that any file format is acceptable as long as there is automatic generation of the output and input converters. This allows but does not require multiple formats.

Access speeds

The different requirements for random and serial access may support the use of more than one file format.

Object references

Object oriented file formats will naturally act as a collection of objects and support associations between these objects. We expect to implement data sharing and inter-EDO references at higher levels (preceding sections) and so the support of associations is not a requirement for the file format. However, if it does exist, we may want to take advantage of it to improve I/O performance. Similarly if the format supports associations between different files, we may want to take advantage of that as well. If such support is provided, it could be used in place of the higher-level mechanism at the high price of selecting a single file format.

Direct access

In may be desirable to study a subset of the data using direct-access data browsers that require use of their native data format. Here we refer to input (and output?) that is done without relying on conversion. Obvious examples are the ROOT object and tree structures. The desire for functional object references may impose further requirements on the browser or the implementation of these references at the file format level.

HES interface

Finally, all file formats must present a common interface that provides the functionality required for the HES view.

Use cases

Athena/StoreGate

The file format is used in Athena/StoreGate with a HES store view. Converters generated with ADL are used to covert data between persistent and transient representations.

Data browser

A native data browser is used to examine a file created with Athena/StoreGate.

Requirements

1. The file format must provide the HES file interface.
2. There must be automatic generation of converters for output and input. These must be based on the ATLAS ADL.
3. Any new file formats should offer some advantage over existing formats. E.g. faster access, better compression or direct access with useful browser.

Design

The HES file interface is described in the previous chapter and ADL is described elsewhere[7]. These can each be implemented as independent layers over the system that provides direct access to the files.

16. ROOT file format

Here we describe a file format built as a layer over the persistency offered by ROOT. There are many ways in which ROOT might be used. We choose one and call it the ROOT file format. It might be better to give this a more specific name such as the BNL ROOT format.

HES is our main use case and leads to the requirements discussed in the preceding two sections.

Use cases

HES

The ROOT file format is used with HES to read and write event data.

ROOT browsing

Files in the ROOT file format are read directly in ROOT possibly with some supporting shared libraries. Event data is browsed including the possibility of following references between objects in different EDO's, which may be in different files.

Requirements

1. The file format must meet all the requirements for AHES file format as outlined in chapter 14.
2. ADL provides the means to generate a transient representation (i.e. the C++ header) for any class defined in ADL. This is the format in which StoreGate holds objects. The ROOT file format defines a different representation, called the persistent representation, in which objects are stored in files. We require a means to automatically generate the persistent representation and the converters between transient and persistent representations from the ADL definition.
3. It should be possible to directly browse ROOT format files using ROOT.

4. When files are browsed in ROOT, it is possible to chase object references into a different EDO and into a different file.

Design

Organization

The code for the ROOT file format includes a base package for reading and writing persistent data objects. This package depends only on ROOT. A second package depends on this and on the HES interface and in particular HesFile.

The base package has a class (call it RootFile) that encapsulates and provides access to the data in a ROOT file. This interface closely reflects the structure of the file. The HES interface package provides a class (call it HesRootFile) that inherits from HesFile and provides access to a RootFile. The interface of HesRootFile is almost completely defined by HesFile and its implementation can be inferred from the interface of RootFile. In the following we focus on RootFile with a few comments about the structure of the file.

File structure

Files include a header and event data. RootFile accesses the header to provide the following:

1. Logical file name
2. Stream type name and definition, i.e. which placement category names
3. Definitions for each of these placement categories, i.e. which type-keys
4. A list of included event ID's including the number of events and the minimum and maximum event ID's
5. The size and checksum for the file
6. The production and production thread ID's
7. The creation time
8. The mapping between type ID (ATLAS class ID) and class name for each type represented in the file
9. The mapping between file ID and logical file name for this file and all those referenced in the data held by this file

The event data includes an entry for each placement category for each event ID. This entry is either a reference (the logical name of the file holding the data) or a collection of EDO entries mapped by type-key. The included type-keys are consistent with the placement category definition in the header.

The entry for each EDO is the address of the persistent representation of an EDO indexed by type-key and EDO ID. This address is null if the EDO is held by reference instead of value.

17. BNL contributions

Magda

Magda is the present implementation of the file location (replica) catalog.

Spider

Spider crawls over file catalogs and verifies their contents. Something similar could be used to verify and construct EDO catalogs.

HEMP

HEMP[3] is a prototype of the relational database part of the hybrid model.

ROOT format

BNL is proposing and implementing a specific ROOT file format. This is described above.

18. References

1. This document and links to some of its referenced documents may be found at http://www.usatlas.bnl.gov/~dladams/hybrid.
2. "ATLAS – Database (ADB) Architecture design document", G. H. Chisolm, E. D. Frank, D. M. Malon, and A. C. Schaffer, ANL/DIS Draft Ver: 0.4 (9/17/2001 8:50 AM).
3. HEMP – Hybrid Event Metadata Prototype, http://atlassw1.phy.bnl.gov/hemp/info.
4. Athena
5. Gaudi
6. StoreGate